CLO2: Develop game systems that integrate multiple components, including AI and system architectures

2.1: Identify core components within game systems (e.g., AI and player interaction)

In this project, the core components include:

- **AI Controller (Enemy)**: Reacts to player proximity and visibility using raycasting and NavMesh pathfinding.
- **Player Controller**: Allows user input (WASD/Arrow keys) to move the player across the map.
- **Environment (Obstacles)**: Objects that impact visibility and pathfinding for AI and offer strategic hiding spots for the player.
- **State Machine Architecture**: The AI transitions between **Patrolling**, **Chasing**, and **Searching** states depending on the player's presence.

These components work together to deliver dynamic interactions between the **player and AI**, forming the core of the gameplay system.

2.3: Design systems that facilitate communication between game components, ensuring smooth interaction and system cohesion

Your AI system demonstrates effective communication between components:

- **AI uses Raycast and Distance Checks** to sense the **player's position**.
- It **transitions between states** (Patrol, Chase, Search) based on real-time conditions like visibility and distance.
- **Player behavior affects AI** (e.g., hiding behind obstacles triggers the "Search" state).
- The use of **shared scripts and references** (like `Transform player`) ensures data flow between the AI and the Player.

This results in **smooth and cohesive system behavior** across multiple components.

CLO3: Apply industry-standard C# programming techniques to build flexible and maintainable game systems

3.1: Implement interfaces and design patterns in C# to create reusable and modular game code

Your AI code demonstrates maintainable design practices:

- **Finite State Machine (FSM)**:
    - Clearly separates logic for each state (`Patrolling`, `Chasing`, `Searching`).
    - Allows for **easy expansion** (e.g., adding "Attack" or "Flee" states later).
- **Modularity**:
    - Patrol logic, detection logic, and transitions are **independently structured** in functions.
- **Reusability**:
    - The AI script is **not hardcoded** to specific players or waypoints—it works with any assigned references.

CLO4: Integrate AI-driven systems to enhance gameplay and player interaction in Unity

4.1: Implement AI behaviors using Unity's tools, such as NavMesh and state machines

In your project, AI behaviors are implemented using:

- **Unity NavMesh & NavMeshAgent**:
    - Handles pathfinding and movement across terrain.
    - Reacts to **obstacles**, **walls**, and **player movement**.

- **Custom FSM (Finite State Machine)**:
    - Built using `enum` and `switch` to manage state transitions.
- **Raycasting & Field of View Logic**:
    - Mimics realistic vision, allowing AI to detect and chase players only when visible.

These are **industry-standard Unity tools** combined with structured C# logic.

4.2: Evaluate the interaction between AI systems and other game systems for balance

You evaluated and tested:

- **Obstacle Interaction**:
    - AI avoids or is blocked by environment objects (walls, cover).
- **Detection Tuning**:
    - Adjusted field of view and detection range for balance — AI shouldn't be too easy or too hard to escape.
- **Player-AI Dynamics**:
    - When the player hides, AI realistically searches last seen position before returning to patrol.
- **Navigation Re-Baking**:
    - Every time you added obstacles, you rebaked the NavMesh to maintain realistic movement.